Load-Balanced LU and QR Factor and Solve Routines for Scalable Processors with Scalable I/O

Load–Balanced LU and QR Factor and Solve Routines for Scalable Processors with Scalable I/O

Jean-Philippe Brunet
Palle Pedersen
S. Lennart Johnsson

TR-20-94

August 1994

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in *Proceedings of the 17th IMACS World Congress*, Atlanta, Georgia, July 11–15, 1994.

Load–Balanced LU and QR Factor and Solve Routines for Scalable Processors with Scalable I/O

Jean-Philippe Brunet, Palle Pedersen and S. Lennart Johnsson
Thinking Machines Corporation
245 First Street, Cambridge, MA 02142

Abstract.

The concept of block–cyclic order elimination can be applied to out–of–core LU and QR matrix factorizations on distributed memory architectures equipped with a parallel I/O system. This elimination scheme provides load balanced computation in both the factor and solve phases and further optimizes the use of the network bandwidth to perform I/O operations. Stability of LU factorization is enforced by full column pivoting. Performance results are presented for the Connection Machine system CM–5.

1 Introduction

Load–balance for in–core matrix factorization on distributed memory architectures can be achieved using a cyclic ordering of the data. In fact, one need not allocate data explicitly in a cyclic fashion. Instead, the elimination can be performed in a cyclic order. That block–cyclic order elimination is an efficient alternative to block–cyclic data allocation for in–core dense matrix factorization was shown in [1]. The present note extends this concept to out–of–core matrix factorization and shows that it further allows for optimum use of the network bandwidth to perform I/O operations.

2 Out–of–core block–cyclic Gauss elimination

The coefficient matrix of size N is partitioned in blocks of columns

$$A = [A_1, A_2, ..., A_l],$$

where each A_i is a block of $M = N/l$ columns, or a panel. We use the standard left–looking block LU algorithm, whereby every panel A_i is first updated by formerly factored panels $A_j, j = 1, ..., i-1$ and then factored. The use of blocks of columns A_i allows for full column pivoting during

the factorization of panels, an essential feature for maintaining stability for general matrices. One penalty for using rectangular blocks of data is that the size of the active data structures changes as the factorization proceeds. The active data structures are the various sections of a panel $A_i(1:N,:), A_i(M+1:N,:), ..., A_i((i-2)M+1:N,:)$, which are updated respectively by the factors of $A_1, A_2, ..., A_{i-1}$, and finally $A_i((i-1)M+1:N,:)$, which is the portion of panel A_i to be factored. After panel A_i is done the upper portion $A_i(1:(i-1)M,:)$ and lower portion $A_i((i-1)M+1:N,:)$ are written to the external storage system along with the relevant pivot information. They will be read later during the solution phase in the forward elimination and backward substitution, respectively.

An efficient implementation of the above schemes on distributed memory architectures hinges upon load–balanced computation while operating on sections of panels, during both the factor and solve phases. Load–balanced computation, which stems from even partitioning of the sections among processors, also entails optimum I/O bandwidth in the transfer of data between processors and the external data storage. Those desirable features are achieved by cyclic factorization of the panels.

Cyclic elimination has been used successfully to perform in–core matrix factorization and solve on a parallel architecture with distributed memory [1]. Blocking was used to further enhance arithmetic performance. Consider the factorization of a rectangular panel A_i. In effect, a block–cyclic permutation $\tilde{A}_i$ is factored

$$\tilde{A}_i = P1^{-1} A_i P_2,$$

where P_1 and P_2 are the permutations giving the correspondence between standard and block–cyclic row order and column order, respectively, for A_i [2]. Given a $p \times q$ processor grid onto which the array $A_i(N, M)$ is mapped, the block–cyclic elimination order selects the first block of rows of A of the first row of processors for the first block elimination, the first block of rows of the second row of processors for the second block elimination, and so on until one block row has been selected from each of the p rows of processors. Then, the second block of rows from the first processor row is selected, etc. Columns are treated similarly. Assuming M is a multiple of bp and bq, where b is the in–core block size, then each of the sections of A_c of the form $\tilde{A}_i(iM : jM,:)$, $i, j = 1, ..., N/M$, is evenly distributed on the $p \times q$ processor array. The same block–cyclic elimination scheme can be used to perform out–of–core matrix factorization efficiently. In effect, a block–cyclic

permutation of the coefficient matrix is factored

$$\tilde{A} = [P_1^{-1} A_1 P_2, P_1^{-1} A_2 P_2, ..., P_1^{-1} A_l P_2]$$

where P_1 and P_2 are is permutation matrices of size N by N and M by M, respectively. The overall effect of cyclic permutation can be expressed as

$$\tilde{A} = \hat{P}_1^{-1} A \hat{P}_2,$$

where $\hat{P}_1 = P_1$ and $\hat{P}_2$ are block diagonal matrices of size N by N with N/M blocks of size M by M all equal to P_2.

Once the matrix is factored, solution of the corresponding linear system can proceed, with multiple right–hand sides, if needed. The lower and upper portions of the panels are read from external storage, along with pivoting information. Thanks to the block–cyclic elimination scheme the panel sections are evenly distributed among the processing nodes allowing optimum use of the network bandwidth during I/O reads. Forward elimination and backsubstitution entail solving block–cyclic triangular systems of equations, which are well balanced operations [1]. Prepermutation and postpermutation of the solution matrix is needed [1]. The only difference with the in–core algorithm is the block structure of P_2, which causes block permutation of the solution matrix in lieu of a generalized shuffle.

In general, one does not expect the dimensions of a given matrix to be a multiple of the dimensions of the processor array it is mapped onto, and the block–cyclic elimination scheme is implemented in such a way as to accommodate all possible cases. In the context of the out–of–core factorization, however, the width of the panels, M, is an adjustable parameter, set internally according to the amount of memory available. Given this degree of freedom, we have found convenient to choose M, p and q (where pq is the number of processors available) such that the panel dimensions should be a multiple of bp and bq. The in–core blocking factor, b, is kept to a fixed value that ensures good performance in BLAS–3 operations (typically 8 or 16). Such a perfect mapping of the panel on a grid of processing nodes also requires choosing N to be a multiple of M. As a result, the matrix actually factored may be larger than the original matrix. The extra bottom rows and rightmost columns are padded internally with the identity matrix.

3 Implementation on the Connection Machine systems

A block–cyclic out–of–core Gauss elimination and solve has been implemented on the Connection Machine systems, a family of distributed memory supercomputers. Here we present performance results for the CM–5. The external storage device for the CM–5 is a scalable array of disks (SDA). A Unix–compatible scalable file system, *sfs*, supports data parallel I/O operations with transfers between processing nodes and the disks occurring through the data network.

The performance and scalability of the out–of–core elimination and solve relies heavily on in–core rectangular LU factorizations, triangular solves and matrix multiply. Those matrix operations are implemented efficiently in the CMSSL (Connection Machine Scientific Software Library) [2]. The out–of–core codes were actually written in Connection Machine Fortran (CMF) [3], a high level language that implements the Fortran 90 array syntax, with calls to CMSSL. CMF further supports I/O operations for arrays. The decision to enforce a perfect mapping of panels on a grid of processing nodes allow us to reference upper and lower panel sections as independent arrays in the global address space using CMF aliasing facilities [4].

Numerical experiments were conducted for double precision complex coefficient matrix with elements chosen randomly between 0 and 1 (in modulus). The machine is a 64 node CM–5 with 32 Mbytes of memory per node and a peak performance of 128 Mflops per node, hence a total performance peak of 8 Gflops. It is equipped with an SDA, the size of which was scaled to the size of the matrix being factored. The matrices factored were of size N=9600, 24576, 51200 and 76800 and the corresponding number of disks were 0 (in-core solution), 32, 64 and 118, respectively. Running time and performance results are summarized in Table 1.

References

[1] W. Lichtenstein and S.L. Johnsson, *Block cyclic dense linear algebra*, SIAM J. Sci. Comp., 14 (1993).

[2] *CMSSL release notes for the CM-200*, Thinking Machines Corporation (1993), 164-194. *CMSSL for CM Fortran: CM-5 Edition*, Vol I, Thinking Machines Corporation (1993), 275-308. CMSSL is the Connection

N	Factor		Solve (1024 rhs)		Total	
	time	perf(Gflops)	time	perf(Gflops)	time	perf(Gflops)
9600	599sec	3.94	258sec	2.93	857sec	3.63
24576	3.2hrs	3.43	0.4hrs	3.31	3.6hrs	3.42
51200	28.2hrs	3.52	1.9hrs	3.10	30.2hrs	3.49
76800	94.6hrs	3.55	4.22hrs	3.18	98.8hrs	3.53

Table 1: Out–of–core block–cyclic LU solver performance for a double precision complex matrix of size N on a 64 node CM–5.

Machine Scientific Software Library, which is available on Connection Machine systems.

[3] *CM Fortran Reference Manual*, Version 2.1, Thinking Machines Corp. (1993).

[4] *CM–5 CM Fortran Performance Guide*, Version 2.1, Thinking Machines Corp. (1993).